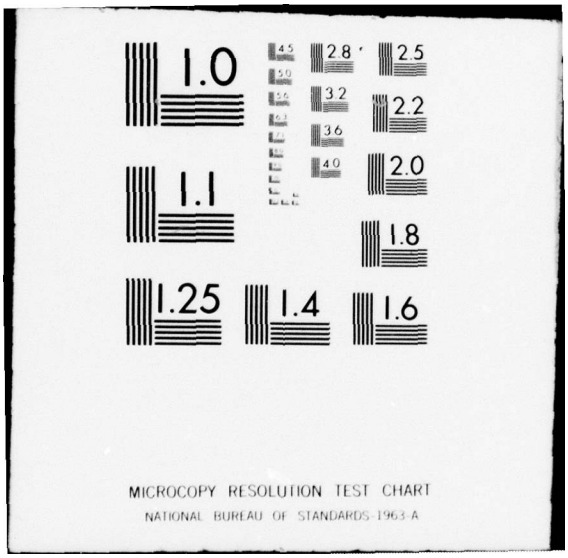
AD-A067 303 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
M-MODULES: A DESIGN METHODOLOGY.(U)
MAR 79 M C PEASE N00014-77-C-0308
UNCLASSIFIED SRI-TR-17 NL

END
DATE FILMED
6-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

M-MODULES: A DESIGN METHODOLOGY

Technical Report 17

SRI Project 6289
Contract No. N00014-77-C-0308

March 1979

By: Marshall C. Pease, Staff Scientist
Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

Attention: Marvin Denicoff, Program Director
Contract Monitor
Information Systems Branch

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SRI-MK- Technical Report 17	2. GOVT ACCESSION NO. (1)	3. RECIPIENT'S CATALOG NUMBER (7) Technical Report 17
4. TITLE (and Subtitle) (6) M-Modules: A Design Methodology		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Marshall C. Pease		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-77-C-0308 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, CA 94025 ✓		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-308
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research (11)		12. REPORT DATE March 1979 / 13. NO. OF PAGES
		15. SECURITY CLASS. (of this report) Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) (12) 18p.		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

modules, modular design, hierarchical design, design methodology, knowledge-based module , knowledge-based system , models, model-driven module, communication condition

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The theory of what we call M-modules for model-driven modules is developed. An M-module is a knowledge-based program module in which the knowledge is encoded as a model that remains available for modification or adaption to meet changing requirements. The methodology of M-modules can help the system designer to make effective use of the decomposition properties of a complex problem. It helps him to decompose an application problem into a hierarchy of subproblems that are much more easily understood and managed by a human user. An important feature that makes the decomposition useful is the methodology orders system operations in a way that prevents deadlock and that contributes to avoiding other system malfunctions. → next page

DD FORM 1 JAN 73 1473 410281

EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

cont.

An M-module contains three components: a model, a set of values, and a set of procedures. The model encodes knowledge about the domain that is the responsibility of the M-module. The set of values describes the M-module's current information about that domain to the level of detail relevent to the M-module. The procedures are those necessary to modify or retrieve information contained in the set of values, or to modify or retrieve the model or the procedures. The M-module accepts information or demands from outside itself--either from a user or from another M-module--and responds according to its internal condition and the constraints of its model.

A key aspect of M-modules is that it leads to a hierarchical system design. The entire system can itself be regarded as an M-module which is composed of a set of lower-level M-modules. An M-module at any level above the lowest can be decomposed into subordinate M-modules plus some auxiliary components. How this decomposition is done is specified rather closely by the theory. The decomposition process is significant since it provides a discipline within which the implications of various components of knowledge can be developed in an orderly manner. The design discipline that results can be understood as a variation and implementation of HDM (hierarchical design methodology) developed at SRI International.

The development of the theory has been based on our experience with the experimental system ACS.1 (the Automated Command Support system). The intent has been to put into general form the design principles and techniques found useful in that system. We believe that those principles and techniques should be applicable in many application environments. The development of an abstract theory of M-modules should lead to a better understanding of when it is reasonable to consider the use of M-modules. The final section of this report seeks to identify the features that make the use of an M-module design feasible and potentially important. The possible advantages of using a design approach based on M-modules are discussed.

ABSTRACT

The theory of what we call M-modules for model-driven modules is developed. An M-module is a knowledge-based program module in which the knowledge is encoded as a model that remains available for modification or adaption to meet changing requirements. The methodology of M-modules can help the system designer to make effective use of the decomposition properties of a a complex problem. It helps him to decompose an application problem into a hierarchy of subproblems that are much more easily understood and managed by a human user. An important feature that makes the decomposition useful is the methodology orders system operations in a way that prevents deadlock and that contributes to avoiding other system malfunctions.

An M-module contains three components: a model, a set of values, and a set of procedures. The model encodes knowledge about the domain that is the responsibility of the M-module. The set of values describes the M-module's current information about that domain to the level of detail relevent to the M-module. The procedures are those necessary to modify or retrieve information contained in the set of values, or to modify or retrieve the model or the procedures. The M-module accepts information or demands from outside itself--either from a user or from another M-module--and responds according to its internal condition and the constraints of its model.

A key aspect of M-modules is that it leads to a hierarchical system design. The entire system can itself be regarded as an M-module which is composed of a set of lower-level M-modules. An M-module at any level above the lowest can be decomposed into subordinate M-modules plus some auxiliary components. How this decomposition is done is specified rather closely by the theory. The decomposition process is significant since it provides a discipline within which the implications of various components of knowledge can be developed in an orderly manner. The design discipline that results can be understood as a variation and implementation of HDM (hierarchical design methodology) developed at SRI International.

Another key aspect is the way communication between M-modules is limited. Communication is permitted only when certain restrictions called the communication condition are met. This restriction enforces order on how an M-module can respond to an external stimulus. The effect is to ensure freedom from deadlock. Furthermore, although the

theory does not yet provide a comperable condition that will generally assure that a stable, self-consistent state is always reached, the use of the communication condition does facilitate designing particular systems that are stable.

The development of the theory has been based on our experience with the experimental system ACS.1 (the Automated Command Support system). The intent has been to put into general form the design principles and techniques found useful in that system. We believe that those principles and techniques should be applicable in many application environments. The development of an abstract theory of M-modules should lead to a better understanding of when it is reasonable to consider the use of M-modules. The final section of this report seeks to identify the features that make the use of an M-module design feasible and potentially important. The possible advantages of using a design approach based on M-modules are discussed.

CONTENTS

ABSTRACT 1

ACKNOWLEDGEMENTS 4

I INTRODUCTION 5

II GENERAL DISCUSSION 10

III ACS.1: AN ILLUSTRATION OF A SYSTEM OF M-COMPLEXES 19

IV IMPLEMENTATION ASPECTS 27

A. PRIMARY AND SECONDARY INFORMATION. 28

B. REPRESENTATION OF PRIMARY INFORMATION 30

C. PASSIVE AND ACTIVE EXTERIOR STATES OF AN INCOMPLETE M-COMPLEX 33

D. EXTERIOR AND INTERIOR OPERATIONS 35

V CONCLUSIONS 39

REFERENCES 41

DISTRIBUTION LIST 43

ACKNOWLEDGEMENTS

Daniel Sagalowicz, who has participated in this research, has contributed significantly to the ideas incorporated in this theory. We also wish to acknowledge the helpful discussions we have had with Jack Goldberg (project supervisor), and with Peter Neumann.

I. INTRODUCTION

This report presents the theory of what we call M-modules, i.e., model-driven modules. This theory has been developed to provide a generalization of the virtual modules of the experimental system called ACS.1 (Automated Command Support) [1]. As the design principles used in that system appear to have possible application to a wide range of needs, the concept of M-modules has been developed to explore this potential.

As described in reference [1], ACS.1 is an experimental system demonstrating the use of advanced techniques in support of managerial decision-making. The system is concerned with the planning of operations in the organization being managed, the maintenance of operational plans when conditions change, and the recording and analysis of completed operations. The context of the study has been the simulated command of a naval air squadron operating from a carrier. The principal operations that require planning are air missions. Other events and activities must be tracked if they can affect the planning function or destroy the workability of an existing plan. The system must maintain up-to-date information on the condition and expected use of all resources (people, equipment, or facilities) that may be needed in an operation. The system's response to exogenous demands for a new plan or for changes to an old one must be consistent with this information as specified or implied by the knowledge encoded in the system.

The main elements of ACS.1 are virtual modules called planners [2] and schedulers [3]. A planner is responsible for creating and modifying plans for a particular type of activity whose description is encoded in what is called a process model. A scheduler is responsible for coordinating the planned utilization of a particular kind of resource. The constraints that define how the resources can be used are encoded in what is called a resource model. The planners and schedulers operate in an essentially autonomous way, each being responsible for its own domain and each responding to all demands that affect that domain. Coordination among the planners and schedulers is obtained by messages transmitted through a central module called the message handler [4].

The theory of M-modules generalizes the design concepts used in the ACS.1 planners and schedulers. A central feature of the theory is that it leads rather naturally to systems that are hierarchically

organized and constructed. A given M-module can often be decomposed into a set of subordinate M-modules plus certain auxiliary components. The subordinate M-modules are said to be at the next lower level. Similarly, it is often useful to combine a set of M-modules at a given level with some auxiliary components to form a higher-level M-module.

The hierarchical system of M-modules is somewhat analogous to the hierarchies of abstract machines used by Dijkstra [5, 6]. The concept is different, however, since it depends on quite different kinds of abstraction. Here a level implies something about the scope and detail of the knowledge used by the M-modules on that level. An M-module on a given level uses all the knowledge of the M-modules on all lower levels. In turn, an M-module contributes its knowledge to any higher-level M-module that contains it. This definition of the system's levels can be regarded as a kind of abstraction process. However, it is sufficiently different from Dijkstra's viewpoint to lead to substantial differences in implementation.

The operation of a system of M-modules is conceptually related to the use of procedural nets within a hierarchy of abstraction spaces as developed by Sacerdoti [7, 8]. Sacerdoti is concerned with planning operations under conditions that make it important to defer planning decisions until as late in the process as possible. He addresses this problem by developing a series of plans, each of which is suitable for a particular abstraction space in which detail is suppressed. A related process occurs in ACS.1. An ACS.1 planner can generate a plan only if it possesses the necessary knowledge. Where additional detail is needed, the planner must request a subplan to be generated by another planner with the appropriate knowledge. The mission planner, for example, is not able to plan the preparation of an aircraft for flight; it knows only that such preparation is necessary. As part of its attempt to create a plan, therefore, it issues a request for a subplan for the task of preparing the aircraft. This request is made of the planner that has knowledge of this task. The transfer of control among the planners and schedulers of ACS.1 as a plan is created is analogous to the shift of control and attention among Sacerdoti's abstraction spaces.

A more immediately useful connection can be made with the theory of finite-state machines [9]. In this view a system of M-modules is seen as a hierarchy of finite-state machines in which a state at one level is chiefly determined as the product of the states of its component machines at the next lower level. (The additional elements that contribute to the state at the higher level can be ignored for the present, they are discussed later.) A state transition at one level is described as mainly the consequence of a set of transitions in the lower-level machines. This viewpoint is useful in interpreting the behavior of M-modules. However, it is of little value in developing the representation of an M-module. In

particular, neither the stable states of an M-module nor its transition rules are given explicitly. A stable state of an M-module is defined negatively as a condition in which none of its model's constraints are violated by any of its values. The transition rules of an M-module are even more obscure, being determined implicitly as the result of inputs that cause the system to change from one stable state to another. The immediate effect of an input can be to make the value set violate some of the module's constraints. When this is so, the module makes changes intended to eliminate the violations. The actual effect of the change, however, can be to introduce new violations that cause additional changes. An extensive sequence of changes may be necessary before a condition is reached that satisfies all the constraints. It is only when this final condition has been reached that the actual transition rule can be said to have been executed.

The concept of M-modules is related to that of Parnas modules [10, 11] but again with a significantly different representation. In particular, an M-module need not contain any function that can be identified as an O-function. (An O-function, as Parnas uses the term, is one that changes the state of the module. It is distinguished from V-functions that return values without changing the state, and from OV-functions that do both.) The M-module contains procedures or functions that can change the values it holds, but the actions of these procedures are not fully defined by the functions. The effect of a function is also determined in part by the model. Further, as discussed above, an input from an external source may initiate a whole sequence of adjustments to the value set before all of the model's constraints are satisfied. The O-functions are only implied as the combined effect of the procedures and the model, and as the total effect of whatever sequence of changes is necessary to achieve conformance to all the model's constraints.

The relation to Parnas modules is significant,however. There is a strong connection between the design principles implied by M-modules and the hierarchical design methodology (HDM) developed at SRI International [12]. To the extent that M-modules can be interpreted as a variation of Parnas modules, the use of M-modules for the design of a system can be understood as a variant form of the general methodological concepts of HDM.

Finally, the concept of an M-module can be compared with the frame theory of Minsky and Winograd [13, 14]. In fact, we regard an M-module as a particular way of representing and using a frame. The knowledge contained in the model of an M-module specifies the relations that define a frame of a particular kind. The procedures of the M-module are those needed to elaborate the general description to fit the particular requirements that may have been given to the

M-module. The information contained in a demand for a particular instantiation can be understood as specifying certain aspects of the frame. The task of the M-module is to use that information to develop other aspects of the frame in accordance with the relations that define it.

We have referred to instantiations of an M-module's model. Just what constitutes an instantiation is itself defined by the model. To illustrate, an ACS.1 planner contains what is called a process model that describes a particular kind of operation in general terms. An instantiation of this model is an operational plan. The process model specifies what information must be included for a plan to be complete. Usually a plan must specify the anticipated start and end times of all tasks in the operation and the names of all resources that are to be used. However, many of these details might be omitted in a contingency plan, left to be filled in when the plan is put into effect. In that case, the process model would indicate which components must be given values and which are to be left open for later completion.

Another significant function of an M-module is to maintain its instantiations. What is meant by this is also defined by the model. Typically, an M-module must be ready to modify its instantiations, as necessary, to keep them consistent with its model and with the information it has been given. For example, a plan being maintained in ACS.1 may have to be changed when new information is received from outside. The plan for a flight will have to be modified if the anticipated pilot is later reported as sick. The maintenance of a plan requires that it be modified to keep it consistently workable in accordance with all currently available information.

Although an M-module "knows" what an instantiation is and how to create one, it may not be able to do so entirely by itself. It may need to have other M-modules create instantiations of their models. For example, a planner may need to get subplans for its tasks from other planners, and to get resources assigned by schedulers. To do this, the planner sends messages to the appropriate modules requesting that they create instantiations of their models to meet the required conditions. The creation of an instantiation in one M-module may cause a chain of actions in other M-modules. The entire set of M-modules is thus integrated into a single dynamic whole.

The fact that creation of an instantiation in one M-module can cause new instantiations to be generated in others could have disastrous consequences. There is the possibility of deadlock if, for example, neither of two M-modules can complete its instantiation until has completed its one. There is also the possibility of unbounded behavior as instantiations proliferate without limit through the system. Finally, if one M-module can force a modification of an

instantiation in another, it is possible that the system could enter a thrashing condition in which instantiations are continually being modified without ever reaching a stable end point. We need to know how these various disastrous situations can be avoided. We do not have complete answers to all of them, although it is possible to specify general conditions under which deadlock cannot occur. These conditions are related to those specified by Dijkstra for the THE system [15]. We have not found similarly general methods to assure that neither unbounded behavior nor thrashing will occur. The development of general conditions that can assure freedom from all kinds of misbehavior remains a problem for further research.

In the following sections, we first present a general discussion of the M-module concept in considerably more precise terms than those used in this introduction. Then, in Section III, we focus on certain aspects of the experimental system, ACS.1, to illustrate some features of the concept and to expand on the significance of some of them. In Section IV we discuss other aspects that seem to be more implementation dependent. This material is included primarily to illustrate the feasibility of these features in a practical system. It is not meant to suggest that the approaches discussed are the only possible ones or the best in all situations. In Section V, finally, we seek to put the concept into perspective. We consider what properties an application environment should exhibit to make it feasible and attractive to use the M-module approach.

II GENERAL DISCUSSION

In this section, we discuss in fairly precise terms the concepts of M-modules and of systems of M-modules. We seek to develop a general framework in which to understand the use of these concepts as an approach to system design.

An M-module, for model-driven module, is a real or virtual knowledge-based module. By calling it a module, we mean to imply a substantial degree of autonomy. It is knowledge-based, meaning that its behavior is largely controlled by knowledge that it contains explicitly. This knowledge describes characteristics of some part of the real world which is the domain for which it has been assigned responsibility. By calling it a model-driven module, we mean that its knowledge is encoded in a model, not simply built into its procedures. The research in which the concept originated is concerned with principles for the design of systems to serve management. An important requirement these systems must satisfy is that the manager must be able to keep them attuned to his needs. This makes it necessary to hold the system's knowledge in a form that is accessible for easy modification by the manager or his delegate. The approach used has been to maintain the knowledge in the form of a model which remains accessible.

The knowledge used by an M-module is encoded in declarative form within the model that is part of the M-module. The model asserts the constraints that must be satisfied by the values contained in the M-module, but it does not itself enforce conformity. The procedures used by the module, acting in conjunction with the model, must create the procedural forms that will actually enforce the constraints. This separation of the declarative representation of knowledge from the procedural forms that use it is a key feature of the M-module concept. It makes it possible for the user to examine the module's knowledge and to modify it as desired.

As indicated, an important feature of the M-module approach is that it leads rather naturally into a hierarchical structure for the system. M-modules can be used as building blocks to form successively larger M-modules. At the highest level the system itself, or the part of it that concerns us, can often be regarded as an M-module. Conversely, it is sometimes advantageous to decompose an M-module into a set of lower-level M-modules plus some auxiliary knowledge, values, and procedures. It is a matter of convenience how far the decomposition process should be pursued. An M-module that is not decomposed, whether it could be or not, is called an elementary M-module.

The formal definition of an elementary M-module is that it is a 4-tuple consisting of a name, a model, a set of values, and a set of procedures:

M-module = (name, model, {values}, {procedures}) (1)

where the braces denote sets. This definition is also applicable to any M-module when regarded as a single entity. That is, (1) describes the components of any M-module when its decomposition does not concern us. If it is not elementary, however, it can also be described in terms of the lower-level M-modules and other components into which it is decomposed. In that case it is defined as a 3-tuple consisting of a name, a set of subordinate M-modules, and a set of auxiliary components:

M-module = (name, {subordinate M-modules}, {auxiliary components}). (2)

In representing an M-module in the form of (2), the model, the values, and the procedures of (1) have been sorted out among the subordinate M-modules and the auxiliary components. The auxiliary components include any knowledge, values, and procedures that are not included in any subordinate M-modules. This material is not merely what is left over from the sorting process; it also includes material that is necessary to coordinate the actions of the subordinate M-modules. For example, the auxiliary components must include the facilities that support and control communications between subordinate M-modules.

When represented in either form, an M-module has a name that must be unique within the set of M-modules that comprise the M-module on the next higher level. The name serves as a virtual address and is used as such by the communication processes of the higher level M-module to which the given M-module is subordinate.

If represented in the form of (1), an M-module has a model that encodes the knowledge constraining the M-module's behavior. In shifting to the form of (2), most of this knowledge is distributed among the various subordinate M-modules, with duplication if necessary, and is encoded in their models. A constraint contained in the knowledge can be assigned to a subordinate M-module if it relates values that may coexist in the value set of the subordinate M-module. Conversely, if the constraint relates values that may coexist in different subordinate M-modules, it must be left in an auxiliary component. As a practical matter, we assign knowledge to subordinate M-modules if we can. We want to push elements of knowledge to as low a level as possible within the hierarchy of M-modules.

If it were possible to distribute all the knowledge contained in an M-module's model into its subordinate M-modules, there would be little reason to define the higher-level M-module at all. The principal value of the hierarchical decomposition is that it permits a specification of how the system is to respond to an exogenous demand or to new information. This specification, the details of which are discussed later, is an important aspect that imposes a partial ordering on the actions taken by the system in response to a need. If there were no knowledge that limited the content of the module as a whole, there would be no need for actions at that level. All actions would be for the purpose of satisfying constraints at higher and lower levels. We assume, in general, that there will always be some residue of knowledge not assigned to any subordinate M-module and that constitutes auxiliary knowledge in the higher-level module.

The set of values in the representation of (1) identifies the current state of the M-module when it is regarded as a finite-state machine. The set contains information the M-module has been given or has generated itself. If the M-module is represented as in (2), most of these values will be distributed among the subordinate M-modules, although some may be retained in the auxiliary value set. The state of the higher level module is the product of the states of the subordinate M-modules and the states of any auxiliary components that contain values.

In passing from the representation of (1) to that of (2), we transfer as much of the original value set as possible to the value sets of the subordinate M-modules. We do not permit duplication of the values in this process; a value is either assigned to a single subordinate M-module or retained in the auxiliary value set. The basic rule is that a value may be assigned to a subordinate M-module if it is accessed only by procedures that are either assigned to that module or contained in an auxiliary component. Again as a practical rule, a value is generally assigned to a subordinate M-module if possible.

The principal contents of an M-module's set of values are those that describe instantiations of the model. To illustrate what is meant by an instantiation, a plan developed by an ACS.1 planner is an instantiation of the planner's process model. In the general case, in which the M-module may serve purposes other than planning, the knowledge in an M-module identifies what is an instantiation of the model; it also specifies which attributes require values for the instantiation to be complete, and from what fields or sets the values must be chosen. By and large, the principal function of any M-module is to create instantiations of its model to meet demands placed upon it and to modify these instantiations when necessary. The source of requests for instantiation and of information can be either outside the system or other M-modules. The set of values of an M-module holds

the existing set of instantiations. It also holds the information needed by the M-module to do its job of creating new instantiations and modifying old ones when necessary.

The set of values of an M-module can include information that is not part of any current instantiation. This information may summarize what has happened, record values that otherwise might be lost when an instantiation becomes history or is modified, or maintain derived information that will be useful in controlling other actions of the M-module or the system. In general, if an M-module maintains information that is not part of an existing instantiation, this information will still be related to its present or past instantiations.

When we shift to the representation of (2), we assume that all values in the instantiations of the model in (1) are assigned to the subordinate M-modules. We further assume that these values become instantiations of the models of the subordinate M-modules. We assume that none of the values assigned to the auxiliary value set are included in an instantiation of the higher-level M-module. This assumption may not be necessary, but it appears desirable.

We have indicated that a value assigned to a subordinate M-module shall not be accessed by any procedure that is not either assigned to that same M-module or is part of the auxiliary set of procedures. The corresponding rule for a procedure is that it can be assigned to a subordinate M-module if it must access only values contained in that M-module. If these conditions are not met, the value(s) and the procedure(s) should be assigned to auxiliary components. These rules complement each other. In addition, we also impose the practical rule that all values and procedures should be assigned to subordinate M-modules whenever possible.

The final component of an M-module defined as in (1) is its set of procedures. The principal group of procedures consists of those that act on the set of values, either modifying them or retrieving information. There can also be procedures that act on the model and even on the procedures. As actually implemented, the procedures are likely to be common to many M-modules and held in a common pool. In this case, the M-modules are virtual entities, but the distinction is not vital.

When an M-module is represented as in (2), the procedures may be assigned to the subordinate M-modules, with duplication if necessary. As indicated above, we wish to assign the procedures to subordinate M-modules whenever possible, minimizing the number that must be held in an auxiliary component. Also as indicated, the general rule is that a procedure can be assigned to a subordinate M-module if it must access only values contained in that M-module. A

given procedure can be regarded as having been replicated and copies assigned to different subordinate M-modules. If so, when it is called it may access only values contained in a single subordinate M-module. The procedure that is called in that case is considered to be a (virtual) copy resident in that module.

This completes the general definition of an M-module, but there is one further distinction that needs to be made. We have not actually asserted that an M-module always exists. It proves to be convenient to regard one as existing only when it is acting as a coordinated unit. At other times when its subordinate M-modules may be acting independently, we say that what exists is only an M-complex. The distinction is trivial for elementary M-complexes; they contain no subordinate M-modules that could act independently. It is only in M-modules that are not elementary that subordinate M-modules, or M-complexes, exist--and the distinction then becomes significant.

To be precise, equations (1) and (2) should be rewritten with M-complex substituted throughout for M-module. An M-module is defined recursively as an M-complex such that:

(a) The M-complex is elementary, or

(b) if it is not elementary, then:

* All its subordinate M-complexes are M-modules, and

* Each of its subordinate M-modules is in a stable state.

The definition may seem to require the higher-level M-module to be in a stable state, but this is not so. The constraints contained in the auxiliary knowledge of the higher-level module may be violated, even though every subordinate M-module is in a stable state. If so, further adjustment is necessary before these constraints are satisfied. Until the required adjustments have all been made, the higher-level M-module remains unstable.

In considering the distinction between M-complexes and M-modules, it is useful to expand the notion of a state in a way that is similar to what is done for finite-state machines. So far we have defined only the stable states of an M-module in which its values conform to all the constraints of its model. However, if a change is produced from the outside which affects one or more of its values, and if this change causes a violation of the model, a series of adjustments may be needed before all the model's constraints are once more satisfied. While this happens, the set of values continues to

violate the constraints of the model. To provide a framework in which to specify and examine the adjustment process, we speak of the process as carrying the module through a sequence of interior states. We say that the process terminates in an exterior state when the module reaches a stable condition in which all constraints are satisfied.

As is done in the theory of finite-state machines, we assume that it is only the exterior states that are observable from outside the M-module or that can be influenced by any exterior event. The interior states are assumed to be unobservable from the outside and to be unaffected by any event external to the module. They merely describe the path by which the module seeks to recover the stability of an exterior state.

If the interior states are unobservable, we might question whether their identification is useful and whether they have any real existence. We could argue that they are merely arbitrary points on the path along which the module passes as it moves towards an exterior state. However, here we are considering M-modules that are also M-complexes and so have sets of subordinate M-modules. It is possible to define the interior states of a non elementary M-module in terms of the exterior states of its subordinate M-modules; indeed it proves useful to do so. To be precise, we define the entire system of states, interior and exterior, of a non elementary M-module as the Cartesian product * of the sets of exterior states of its component M-modules. Its interior states are then the members of this product that are not exterior states. The M-complex is not an M-module if it is not in either an exterior or interior state.

An elementary M-complex is regarded as always being either in an interior or exterior state, and so is always an M-module. In this case, we are not much interested in the details of its interior states. An elementary M-module can be regarded as a dynamic entity that can exhibit externally only its stable configurations.

---

* The Cartesian product of two sets is defined as follows: Let the sets be S with members (s1, s2, ..., sm) and T with members (t1, t2, ..., tn). The Cartesian product of S and T is the set of all ordered pairs (si, tj) where si is any member of S and tj any member of T. The generalization of this concept to a finite number of sets is evident.

The concepts that have been introduced enable us to define what we call the communication condition limiting communications within a system of M-complexes. This condition can be stated follows:

(a) All communication occurs between M-complexes that are subordinate M-complexes in a single M-complex, and

(b) No communication occurs within a non elementary M-module unless all its subordinate M-modules are in exterior states.

To make clause (a) generally applicable, we can regard the system and its user(s) as a single M-complex, of which the system is a subordinate M-complex.

Clause (a) limits the whole idea of communication to within non elementary M-complexes. However, the whole system is organized into a hierarchy. Any communication that is needed can be regarded as conforming to clause (a) at some level in the hierarchy. This clause is not really restrictive.

Clause (b), on the other hand, is rather restrictive. It specifies that communication can occur within an M-complex only when it is an M-module. This is reasonable if we recall the intention expressed previously to limit the notion of an M-module to the times when the M-complex is acting as a coordinated unit. But it does sharply limit the times at which a given communication can occur.

The communication condition has wide consequences. It imposes order on the way in which an M-complex approaches a stable configuration that satisfies all the constraints of its knowledge. Specifically, it implies that all adjustments of the values contained in the M-complex occur in bottom-up order. That is, if adjustment to the current set of values in an M-complex is needed, it first occurs at the lowest level at which a model constraint is violated. If necessary, adjustments are made first at the level of the elementary M-modules. The constraints of the next level up are considered only when all values have been brought into full conformity with the bottom-level constraints. Furthermore, if an adjustment at some higher level causes a change that violates a lower-level constraint, then control must return to the lowest level at which a violation exists until it has been cleared. The rules and constraints are applied in an orderly and systematic way, working upward from the lowest level.

Since the entire system, or at last the part of it that concerns us, can be regarded as an M-complex, the ordering of the process of adjusting to new information or demands applies throughout the system.

It is worth examining this ordering property a little more carefully. Consider an M-complex, X, with subordinate M-complexes, A, B, and C. Suppose the value sets of A, B, and C are all consistent with their separate models so that all are in exterior states and X is an M-module. These value sets, however, may not be consistent with an auxiliary constraint of X. X may therefore be in an interior state. This inconsistency is discovered by communication among A, B, and C, and it is recognized that a change is needed to correct the violation. Suppose this change alters a value in A. Suppose this altered value violates knowledge contained in A. By definition, A is now in an interior state and X is no longer an M-module. The knowledge contained in X's auxiliary components cannot now be applied. This situation persists as long as A remains in an interior state; X resumes its efforts to obtain consistency with its auxiliary knowledge only when A has recovered full internal consistency. Control remains at the lowest level at which a constraint is violated.

The communication condition creates the possibility of asynchronous operation. It avoids what otherwise would be severe problems of synchronization. It prevents communication at times when the influence of one M-module on another might be disruptive. An M-module will receive a communication only when it is in an exterior state after having finished made all adjustments that it previously recognized as needed. The impact of the communication can never get confused with the internal process of adjustment.

The most direct way of enforcing the communication condition appears to be through messages and a message handler. A message, as we use the term, is an encoded sequence that contains all the information needed to specify a desired action by the target module. It may be a request for action, asking for an instantiation of the target module's model that will meet certain conditions. It may be a request for information, specifying what information is needed and by whom. In general, a message is a request for a specified action that the originator wants. While we may regard it as a demand, the originator does not know that the requested action will be done. The action that actually results from the message is determined by the state of the module that receives it. Furthermore, a message is not sent directly to the intended receiver, but rather to the message handler. The message handler is a buffer that is an auxiliary component of the M-complex. The originating module can send a message to the message handler at its own convenience. The message is accepted by the target module only when that module is ready to receive it--i.e., when the module is in an exterior state. There is, therefore, no need for any mechanism that will actively synchronize the two modules.

As described above, the message-handling procedure violates the rule that all subordinate M-modules must be in exterior states before any communication is allowed. In ACS.1 full conformance to the rule is an indirect consequence of the fact that the host system is a sequential one with no parallelism. In the more general case, it would be simple to block the message handler from delivering any message until all M-modules in the complex have indicated their readiness.

The use of a message handler is not the only way of achieving the desired effect. Another technique allows module A to enter data at any time into module B, but in way that module B can ignore as long as necessary. In particular, the data is entered into module B as the value of a property that is not examined until module B enters an exterior state. Module B can be seen as providing its own buffer which it ignores until it is ready to be influenced. Again literal observance of the rule that all subordinate M-modules must be in exterior states would necessitate some additional control features, but their design would not be difficult.

One of the important implications of the communication condition is that it provides automatic prevention of deadlock. This is true, providing that (a) the elementary M-modules in the system can not deadlock when acting in isolation, and (b) the communication condition is imposed on each communication as it occurs. If a message handler is used, clause (b) means that an M-module picks up and responds to one message at a time. If the message to which the module responds throws the module into an interior state, then this condition must be cleared before the module can respond to any other message. This prevents the M-module from being thrown into a deadlock by contradictory messages; if a set of conflicting messages exists, the module still responds to them one at a time, completing its response to each before even looking at another. If we assume that each elementary M-module is free from deadlock, the entire part of the system that is an M-complex will likewise always be free from deadlock.

This argument does not assure that the system will not thrash, enter an infinite loop, or generate some other unacceptable action. We have assumed that, whenever an M-module is thrown into an interior state by some stimulus from outside itself, its procedures will eventually lead to a new exterior state. However, there is no certainty that an M-module will never enter an unbounded sequence of interior states. The precise determination of conditions that are sufficient to prevent the possibility of an M-module entering an unbounded sequence of interior states is a subject for further research.

III ACS.1: AN ILLUSTRATION OF A SYSTEM OF M-COMPLEXES

In this section, we use the experimental system, ACS.1, to illustrate the use of M-complexes.

The principal components of ACS.1 are the modules called planners and schedulers, as detailed in references [1-4]. A planner is a unit that knows how to plan an activity of a given type, such as an air mission or the preparation of an aircraft for flight. A scheduler knows how to schedule resources of a particular type, such as the pilots, the aircraft, the launch catapult, and so on. All coordination between planners and schedulers is by messages passed through a unit called the message handler. The entire system of planners, schedulers, and the message handler can be seen as an M-complex--of which the planners and schedulers are subordinate M-modules or complexes. The message handler is an auxiliary component of the M-complex that constitutes the system.

The model used by a planner, called a process model, describes the activity that is the planner's responsibility. For example, the mission planner of ACS.1 identifies a mission as consisting of five tasks, PILOT-BRIEF, A/C-PREP, FLT, A/C-SERV, and PILOT-DEBRIEF, plus two assignments, a pilot and an aircraft. It knows the ordering constraints that apply to these tasks and assignments. It also knows how to plan the tasks if it can do so itself, or how to ask for a plan from another planner when this is necessary. For example, it knows that the task PILOT-BRIEF has a set duration and so can be planned immediately once a start or end time is known. The task A/C-PREP, on the other hand, is planned by another planner that knows it is composed of a fueling, an arming, and an inspection subtask; that other planner also knows that the assignment of a maintenance person is needed. The mission planner knows how to request a plan for the preflight preparation of an aircraft.

The model used by a scheduler, called a resource model, identifies how resources of the given type can be used. The pilot scheduler understands that a pilot can be assigned to a mission, can be resting, can be on sick leave, can be on leave from the squadron, or can be busy in classroom training. It also has rules that govern the way an anticipated use of a given resource can be changed. For example, a pilot that is on leave away from the squadron cannot be assigned to a mission. On the other hand, a pilot who is resting can be assigned if it is necessary, but he should not be if it can be avoided.

The principal values held in a planner are the plans it is currently concerned with. A plan, which is an instantiation of the planner's process model, comprises a specification of the key times and assignments. Normally this means the specification of the start and end times of each task, as well as the name, start, and end times of each assignment. Some of this information could be omitted under certain circumstances. For instance, in contingency planning certain values might be omitted on the assumption that they could be filled in when the plan is implemented. Specified in the model are those properties that must be given values for a complete instantiation.

The set of values held in a scheduler describes all the available information about the given type of resource that might affect its future use. For example, the pilot scheduler not only records what assignments have been made for each pilot, but also such conditions as sickness, leave, training exercises, and so on. Each such entry is an instantiation of the resource model. These conditions are recorded primarily because of their possible effect the availability of a pilot for flying a mission.

Schedulers and planners are quite different in appearance. In the abstract, however, they are quite closely related. Each has a model encoding knowledge that constrains what can happen to its set of values. Each has a set of values comprising its current set of instantiations. Each has available to it a set of procedures that act upon the set of values in accordance with the constraints imposed by the model. Each is an M-complex.

The planners can be decomposed into smaller M-modules or complexes. So can the schedulers, but, since it is not clear that this is helpful, we concentrate on the planners. A subordinate M-module of a planner handles a single task or assignment. Its model includes the constraints that apply to that task or assignment and which relate values that are all part of an instantiation of that one task or assignment.

The value sets of the subordinate M-module of a planner have as their main content the current instantiations of the task or assignment being handled by that M-module. That is, they include the values of the properties that are specified by the model as requiring values. The value sets also include any other entries that can be conveniently associated with a single task or assignment. As an example, a priority index has been associated with each property in a task plan in ACS.1. This index is used to indicate whether the start or end time is to be changed if replanning becomes necessary. The index can be directly associated with the value of the start or end time it controls.

The planner also contains auxiliary constraints. For example, the ordering constraints that tell which tasks must be completed before others can be started are auxiliary ones, because they relate valueisn different tasks. The auxiliary knowledge also specifies or implies which of the subordinate M-modules must contribute complete instantiations before the planner can complete its instantiation.

Communications between the subordinate M-modules in a planner do not use the device of a message handler. Instead, use is made of special property-value pairs to put the interactions of these modules in proper order. For example, suppose the process model of the planner specifies that task A must be completed before task B can be started. As soon as the end of task A has been given a value, it is entered into task B as the value of the property EST for earliest start time. The subordinate M-module for task B pays no attention to the property EST as long as it is in an interior state. It looks to see whether it has a value of EST only after it has reached an exterior state. At that time, if it also has a value for its start time, the module compares the start time to the value of EST to see whether they are consistent. If they are not, it knows that replanning is required. On the other hand, if task B has no start time, the value of EST is entered as the start time. The point is that the subordinate M-module uses the property EST as a buffer for incoming information. The value, if any, of EST is used only when the module is in a exterior state in which it can let itself be influenced by information from outside itself. Similar use is made of each of the properties named LST (latest start time) and EET and LET (earliest and latest end time). These properties are used in a way that implements the communication condition.

Other details of the system and its planners, schedulers, and message handler have been given in some detail in references [1-4]. What has been given is sufficient to illustrate the basic concepts. Before we complete this discussion of ACS.1, however, it is worth using the example of ACS.1 to introduce two properties whose presence or absence in a given M-module is an important determinant of the module role and behavior. These are the properties of simplicity and of having what we call a complete model. The effect of the presence or absence of these qualities is illustrated by the obvious differences between the ACS.1 planners and schedulers.

An M-module is simple if none of the constraints expressed in its model or implied in its procedures relate values in different instantiations. Its model is complete if the M-module can always develop an instantiation of its model without using information from any module other than the source of the demand to which it is responding, i.e., without having to consult any other M-module. The

ACS.1 planners may need to consult other planners, but their models do not place any limitations on what values can be simultaneously present in different plans. A planner is a simple M-module with an incomplete model. The opposite is true of the schedulers. An ACS.1 scheduler never needs to call on another planner or scheduler, but its model does place limitations on what values can be simultaneously present in its instantiations. A scheduler is not simple, but has a complete model.

We consider the significance of simplicity, or non-simplicity, first. The simplicity of a planner means that its model never puts limitations on the values in the plans that coexist in it. Clearly, plans can be inconsistent; for instance, two plans in the mission planner that call for the same aircraft simultaneously are inconsistent. However, this inconsistency will not be discovered by the mission planner as the knowledge it utilizes does not specify that this cannot happen. Preventing conflicts of this sort is the responsibility not of the planner, but of the aircraft scheduler.

In any system or subsystem that is an M-complex, possible interactions or conflicts between instantiations are recognized only in its non simple subordinate M-complexes. In ACS.1 the only kinds of conflict that are considered are those that arise from competition for limited resources. The constraints that prevent conflicting assignments of limited resources are contained in the schedulers. Therefore it is only the schedulers that are non simple.

The effect of simplicity is that it sharply limits the information an M-complex must retain. In general, a simple M-complex need retain only those instantiations on which it is actively working, while all others can be stored in a data system or elsewhere, as convenient. An ACS.1 planner needs to retain only the plans it is currently working on, if it also has the ability to retrieve a plan that needs re-evaluation or replanning. It can, of course, retain other plans if it is desirable and convenient to do so, but it does not need to. A non simple M-complex, on the other hand, must retain all of those instantiations with which any possible new instantiation could interact. The ACS.1 schedulers must retain all available information about all anticipated uses of its resources. Any of this information could restrict its ability to respond to a future request for assignment, and so must be kept available. The number of instantiations a scheduler may need to retain can be very large--this is an immediate consequence of its non simplicity.

Moving now to the notion of a complete model, we observe that the completeness of the resource model in a scheduler means that a scheduler's response is always based entirely on information it already possesses. This is so whether it fulfills the request exactly as wanted, offers an alternative which is not exactly what was requested, or refuses the request. It never interrogates any other module. In contrast, since a planner may have to ask another planner for a subplan, or ask a scheduler for an assignment, its model is not complete.

The completeness of the model of an M-complex has a practical effect in the avoidance of thrashing and other misbehavior. An ACS.1 scheduler, which is always both complete and non simple, can determine quickly and unambiguously what options are available in responding to a request. These options stem from information contained entirely within the scheduler and fully available to the procedures that determine the options. Since the scheduler does not need to develop tentative responses that must then be evaluated and perhaps revised, there is no way it can thrash.

On the other hand, an M-module with an incomplete model can misbehave in several ways. Consider how it responds to a demand for a new instantiation. It starts by sending requests for information or action to other modules. The responses can be regarded as defining a set of options for the M-module. To select an option, it evaluates the responses according to its own information and goals. This evaluation may be lead to the issuing of new requests for information or action that are different from their original counterparts. The responses to these latest requests define a new set of options. There is no a priori reason why, on evaluating these new options, the M-module cannot be made to develop new requests that are, in fact, identical to its original set. If so, the M-module is in an infinite loop. Similarly, if the new set of requests are always different from any that have been sent before, there is still no guarantee that the process will ever converge to a solution in which all the parties involved can concur. The system can thrash.

The problem is that an incomplete M-module depends in part on its own information and in part on information contained in other M-modules. As the latter is not visible to the original module, the module can only request information or action. These requests are dependent on certain implied assumptions about the nature and location of that information. If the assumptions are wrong and the responses it receives do not lead it to revising the assumptions correctly, the result can be pure confusion. The situation becomes analogous to two people who must negotiate with each other, yet who have deep misunderstanding of each other's knowledge, goals, and capabilities.

There is no guarantee that an M-complex with an incomplete model will not thrash or otherwise misbehave. The problem appears to be manageable, however, if the incomplete M-modules are simple--as is the case with the ACS.1 planners. The simplicity of a planner means that it is not required to coordinate information it receives from its requests with much resident information. The information in the replies must be integrated with information in a single instantiation, but all other instantiations can be ignored. If an M-module were both non simple and had an incomplete model, the coordination might require changes in several instantiations. The fact that several instantiations may be affected makes it more likely that the changes will alter the basis underlying the original requests for information or action. The opportunities for trouble can multiply rapidly. In a simple M-module we cannot lay down explicit design principles that will guarantee the absence of trouble, but experience demonstrates that, for all practical purposes, the problem is manageable in ACS.1-like systems.

This observation raises an interesting and important question. Is there ever a need for M-complexes that are either simple and use complete models, or non simple and use incomplete models? In ACS.1, the schedulers always use complete models but may be non simple. The planners are simple, but may use incomplete models. Is there ever need for other kinds of M-complexes? We believe not, but have not yet established the conclusion generally.

It is easy to see that there is no need for M-complexes that are both simple and complete. This follows because a simple and complete M-complex can always be absorbed, with duplication if necessary, into the M-complexes that can call it. Suppose P is a simple planner--i.e., it never calls another planner or a scheduler. Suppose it is called by another planner, Q. When called by Q, P must have all the information it needs to respond. Any part of the needed information not given it by Q must have been already present in P. The information held in P's model and set of values that P used to generate its response could be transferred to Q. If this were done, then Q could generate the same response itself. Q would not need to call on P at all. P could be absorbed into the planner that calls it.

If another planner, say R, can also call P, the critical question is what information in P may be needed by both Q and R. In ACS.1, P uses a process model that adds detail to some task of the process models of Q and R. Each instantiation in P is the result of a request from some planner that is taking a coarser and wider view of the process. Since each instantiation in P derives from a requirement in some other planner, say Q or R, it continues to be the concern only of the planner that set up the requirement in the first

place. Hence an instantiation in P can be associated with the planner, Q or R, that caused it to be created initially. If we now want to absorb P into the planners that can call it, Q and R, we must replicate P's model and procedures, putting copies of them into both Q and R. The instantiations of P, however, can be sorted and each transferred to either Q or R, never to both. P can be absorbed into the M-complexes that can call it. Only its model and set of procedures may need to be replicated, never any of its values.

There is a slight difficulty that we have glossed over. The set of values of P may include values that are not part of any instantiation. These values can record, for example, cumulative figures for the activity described by P. It may not be possible to assign these values exclusively to either Q or Q'. However, such values are maintained either for the sake of convenience or as control parameters. While they are often useful, they are not essential to the concept of an M-module or of a system of M-modules. Even the control properties can be handled in other ways, splitting the required control function between Q and Q'. It can be quite inconvenient to eliminate P as a separate M-module, but this does not affect the argument that it is possible to do so.

It should be emphasized that the argument given does not deny that it may be desirable to use M-modules that are both simple and have complete models. The argument says nothing about the practical advantages or disadvantages of doing so. It only asserts that the possibility exists of absorbing such an M-module into the M-modules that can call it.

In the other case, a non simple incomplete M-complex, we have not developed a similar argument. We believe it is always possible to split the complex into two M-complexes, one simple and incomplete, the other non simple and complete, or into a planner and a scheduler. This appears to be always possible, but it is not entirely clear that we have considered all situations.

To illustrate how a non simple M-module with an incomplete model can be split, consider the following example. Suppose, in ACS.1, we want to plan missions so that there are never more than N aircraft in the air at once. More than N missions can exist concurrently if some are in their preflight or postflight phases. The limitation is to apply only when the aircraft are actually in the air, i.e., when the missions are in the FLIGHT task. If we impose this condition directly, it becomes a constraint among the instantiations of the mission planner. This makes that planner non simple as well as incomplete. However, the condition can be handled differently. We

can set up a scheduler for a dummy kind of resource, say tokens, and give it N labeled tokens. At the same time, the process model of a mission is modified to require the assignment of a token concurrently with the task FLIGHT. Since there are only N tokens, there can be only N concurrent flights. The mission planner, which originally was a non simple M-complex with an incomplete model, has been split into a simple but incomplete planner and a non simple but complete scheduler. It seems plausible that this can be done in all cases, but the possibility has not yet been proved generally.

IV IMPLEMENTATION ASPECTS

In this section we consider other aspects of the M-modules that are more implementation oriented and which may be more dependent on a particular application area.

There are four topics addressed in this section. First, we consider what information may be included in the set of values of an M-complex. This information is categorized as primary or secondary. The primary information includes the values that are the current instantiations of the model. In the second subsection we discuss some of the structural aspects of the primary information, as well as the characteristics of the format in which it can be encoded and which will facilitate the operations of the M-module. Third, we define what we call passive and active exterior states for an incomplete M-complex, and discuss the significance of this aspect and how it can be implemented. Finally, we consider the operations that actually make an M-module a dynamic entity capable of maintaining its own integrity according to the knowledge contained in its model.

A. Primary and Secondary Information

We have briefly mentioned that the set of values of an M-complex can include not only its current set of instantiations, but also other information. Here we expand somewhat on this statement, suggesting the kinds of additional information that may be needed or desired. We are led to categorizing the information held in the value set of an M-module as primary or secondary. An approximate definition is that the primary information includes the current set of implementations and any other information that can conveniently be included with that set. Everything else is secondary.

The primary information that is not part of an instantiation is information that can be usefully associated with one. A simple example from the application environment of ACS.1 is the code indicating the purpose of a flight. This code can be attached to the plan for the flight, even though it is not an essential part of the plan. An example of information that cannot reasonably be associated with a single instantiation, and which therefore is regarded as secondary, is the cumulative total of flight hours for a pilot or an aircraft.

The decision as to whether information is primary or secondary can be made on pragmatic grounds. If we can guarantee the presence in an M-module of a certain instantiation whenever a particular piece of information is wanted, we can attach the information to that instantiation and call it primary. Otherwise the information must be treated as secondary. There is no plan for a flight that will definitely still be active whenever the cumulative flight hours are be needed. Therefore the flight hours constitute secondary information.

The labeling of certain information as secondary does not preclude its being important or having a major effect on the system. For example, the cumulative flight hours for each aircraft can be maintained as part of the secondary information of the aircraft scheduler. This total can be used to control the planning of scheduled aircraft maintenance actions. The cumulative total is maintained in the scheduler, that is where the source information is located. Its effect, however, is to control a major planning operation of the system.

The information held by an M-module can be given to it from an external source, it can be developed as part of the process of generating or maintaining instantiations, or it can be derived from other information the module contains. The information received from outside is accepted because it can influence the process of creating

or maintaining instantiations. Thus, all the information held by an M-module is concerned with the instantiation process in one way or another. However, the connection of a given piece of information with the instantiation process may not be simple or direct. There can be many reasons why a given piece of information cannot be associated usefully with a particular instantiation, and so must be considered secondary.

The secondary information can serve a wide variety of purposes. The question of how it should be stored, or in what format, depends on its nature and the particular purpose for which it is intended. Hence, there is little we can say about its structure or format that will be generally applicable. How it is handled must depend on its particular content and on when and how it can be used.

On the other hand, the primary information exists, first and foremost, to serve the M-module's function of generating and maintaining instantiations. Any other use that may be made of it will be subordinate to this purpose. Its main content is also specified; it is the set of instantiations. The nature and principal use of the primary information is, therefore, rather narrowly specified. Therefore, we can argue that its form should exhibit certain structural features. This viewpoint is pursued in the next subsection.

B. Representation of the Primary Information

The primary information performs a very limited role. To facilitate its utilization in this role, it is desirable to structure its representation in a way that is convenient for that purpose. The particular content of the primary information makes ths possible. The structural features of its representation that are intended to meet this requirement are the following:

1. The set of primary values is divided among a set of labeled partitions with all the primary information about a single instantiation being contained in one partition.

2. Within each partition each value is associated with a "location" specified by the partition and by one or more parameters. The values of the parameter(s) may be numeric, within prescribed limits, or drawn from a specified set of names.

3. The basic mechanism used for the entry or retrieval of primary values is through their locations which thereby act as virtual addresses. These addresses are used by the main procedures for entering, modifying, or retrieving information. Other, more complex, procedures can be built from these basic procedures by, for example, specifying a content-addressed search over a range of locations.

The partitions are chosen so that the constraints of the model always relate values within a single partition. Within this condition the partitions are made as small as possible.

The partitioning of the primary information limits the range of any search procedures that may be necessary to determine whether the set of values obeys the constraints of the model. If a change is made in any value it can introduce constraint violations, but only within the partition holding the changed value. Any checking needed can be restricted to this partition.

To illustrate, each plan held by an ACS.1 planner has its own partition. This is so because a planner is simple and no constraint of its process model relates values in different plans. In general, in any simple M-module we create a partition for every instantiation. The ability to do so is directly due to the M-module's simplicity.

In a non simple M-module, the situation is more complicated. What partitions, if any, are used depends on the nature of the model's constraints. In the schedulers of ACS.1, for example, we create a partition for each resource. The constraints of the resource models used in the schedulers do not relate values for different resources. It is possible to partition the representation of the primary information of a scheduler among its particular resources. In a non simple M-module with a more complex model, a different way of partitioning the primary information might be necessary. At worst, no partition meeting the condition might be available. If so, it is just an inconvenience, not a disaster. The partitions are useful in that they improve the efficiency of the constraint satisfaction process, but they are not essential.

In the ACS.1 planners each plan is a partition labelled by a global identification number. Within a partition the location of a particular value is specified by a list consisting of, first, the name of a task or resource; second, either TASK-PLAN or RESOURCE-PLAN according to whether a task or an assignment is involved; and, third, the name of the property being given a value--START, END, or, if a resource, NAME. This list identifies a location within a plan.

In the schedulers of ACS.1 each partition is identified by the name of a resource. The location within the partition is parameterized by time and restricted to the future. (A scheduler in ACS.1 is not concerned with past events.) It may be noted that this is an inverted system. In the instantiation itself, time is a value, not a property. However, it has proved convenient to use time as the independent parameter in the representation of the primary information.

The choice of a direct or inverted structure is a consequence of the particular constraints of the model. In a simple M-module such as a planner, all applicable constraints relate the values of named properties. In such a case, it is convenient to use a direct structure that can support a call-by-name procedure. (In the planners of ACS.1, this parameter is expanded into a list of labels, but this is a minor variation.) In non simple M-modules such as a scheduler, there can be other kinds of constraints. In the schedulers of ACS.1 in particular, a conflict arises when we try to assign a given resource to different uses at overlapping times. This makes it convenient to be able to retrieve information about the use of a given resource at a given time or over a given range of time. It suggests putting emphasis on call-by-value procedures, since time is the value of the relevant properties.

In summary, it is possible and desirable to organize the representation of the primary information in a way that matches its use in creating and maintaining instantiations. This suggests, first, that the primary information should be distributed among a set of partitions whose boundaries are determined by the applicable constraints. Second, within a partition, the location of a given item of information should be specified in a way that makes it easy to test the applicable constraints. This consideration may dictate that the location be specified by the name of the property being given a value or by the value to which the property is attached. The use of the property name as the locational parameter appears to be convenient in simple M-modules. The use of the value as the locational parameter can be more convenient in non simple ones.

C. Passive and Active Exterior States of an Incomplete M-Complex

It has been found necessary to distinguish between what we call the active and passive exterior states of an M-module with an incomplete model. An active state is one in which the module can initiate action elsewhere in the system; a passive state is one in which the module is precluded from initiating action elsewhere. The distinction applies only to M-modules with an incomplete model, as these are the only M-modules that initiate action elsewhere at all.

The necessity of the distinction arises because the M-module can be waiting for replies to requests it has previously initiated. If it is not precluded from originating new requests while it has outstanding ones, confusion can result.

The distinction needs to be made with respect to each of its instantiations. To be precise, we say that an instantiation is in a waiting mode if the M-module has previously issued one or more requests for action for which no acknowledgement or answer has has been received. Otherwise we say the instantiation is in a filled condition. The M-module, when in an exterior state, is also in a passive state if none of its instantiations are in a filled condition; otherwise it is active. If the state is passive, the M-module is prevented from requesting any new information or action from another M-module. If the state is active, it is allowed to originate new messages about those of its instantiations that are in a filled condition.

A means of keeping track of this distinction is essential to control the behavior of an M-module and to prevent certain types of misbehavior. Without it an M-module may not be able to know whether it has already asked for some action and can therefore be led to issue a duplicate request for some needed action. For example, consider an ACS.1 planner. Suppose it has requested several assignments for a plan it is creating. It is reactivated when it receives a response to any one of these requests. If the power of the planner to originate requests is not inhibited, the planner will discover that it is lacking the other assignments and will issue new requests for them. Since these new requests will not be recognized by the schedulers as duplications, they will lead to duplicate assignments. It is necessary to take some sort of special precautions to avoid such errors.

It must be possible to determine whether a given state is active or passive by means of information contained in the value set. There is no other way an M-module can maintain control information.

The control information, however, must be added to the value set for the specific purpose. This can be done easily. All that is necessary is to add to each instantiation a property whose value is the count of the outstanding requests for actions--each instantiation is filled if the count is zero; otherwise it is waiting. It is then simple to ascertain whether the state of the M-module is active or passive, and to control its operations accordingly. The problem, therefore, is easily handled; it is mentioned here only because it is easy to overlook the need to make provision for it.

We have tacitly assumed that the M-module is simple as well as having an incomplete model. Otherwise we cannot treat its instantiations independently, deciding separately for each instantiation whether it is filled or waiting. As we have indicated before, we believe that a non simple M-module with an incomplete model can always be divided into two M-modules: a simple one with an incomplete model, and a non simple one with a complete model. If this is true, then the control concepts developed here can always be applied.

D. Exterior and Interior Operations

The model of an M-module contains its knowledge in declarative form, only asserting the constraints and requirements that are needed. The M-module must also contain some means to enforce automatically these constraints and requirements automatically. It must be made into a dynamic entity that can act on its own initiative to correct violations of its knowledge. In this section we discuss the mechanics of doing this.

It is convenient to speak of operations as distinct from procedures. To be precise, an operation is a map of the set of values of an M-module into a new set or the same one. It is the action by which a named procedure is called and given the values it needs for its variables and parameters. On the other hand, a procedure is independent of the situation. It exists permanently as a body of code waiting to be called. An operation is an event that occurs at a particular time and under particular circumstances. In general, an operation will involve the execution of an existing procedure.

We distinguish two kinds of operations. Like the two kinds of states, we call one exterior, the other interior. An exterior operation is initiated by an event occuring outside the M-module. The usual event is the arrival of a message requiring that a named procedure be executed with certain values of its variables. Since an M-module cannot be influenced by any event external to it unless it is in an exterior state, exterior operations can be initiated only in such a state. In contrast, an interior operation is usually initiated when the M-module enters into an interior state. It is the means used by the M-module to execute transitions between interior states, or from an interior state to an exterior one.

The initiation of an interior operation is not a simple thing. It must be triggered by the entrance of the M-module into an interior state. The problem is to provide a mechanism that will discern this fact and then initiate an action that is appropriate to the constraint violation that is present. Yet the knowledge that is used by the M-module neither identifies the interior states directly, nor does it specify what state transition should occur from one.

We could consider using the approach of Hewitt and others based on what are called actors [14]. An actor is a procedure with one or more preconditions. The simplest way of using them is for the system to examine its entire pool of actors, form a set of those whose preconditions are satisfied, select one of this set, let it execute its code, and then start over. (There are more complex ways employing

purposeful means to avoid having to examine all actors each time. It is also often possible to be clever in choosing which actor is to execute its code. These refinements do not concern us here.) If we chose to apply a similar approach, we could develop a set of actors, each of which uses as its precondition a test of the set of values against some constraint of the model. If the precondition were found to be satisfied, this would mean that a particular constraint was being violated. The actor itself could make changes in the value set that would eliminate the violations discovered.

This would be a possible approach, but it becomes a difficult one if there are many values in the existing set and many constraints, expressed or implied, to be satisfied. Furthermore, we believe that the nondeterminacy in the selection process that chooses the particular actor to be executed might cause difficulty. It might make it more doubtful if the process could be kept stable.

An alternative method, the one used in ACS.1, regards an interior operation as composed of two parts. The first part uses what are called demons, each responsible for recognizing when a particular constraint is violated in a particular value. A demon is created long before the need for it arises and exists passively as long as the constraint for which it is responsible is not violated. When the value for which a demon is responsible changes, the demon checks to ascertain whether a violation of its constraint has been introduced. If so, the demon is said to have been awakened. It then executes the second part of the interior operation, which is the call of a procedure named by the demon. The procedure specified in the demon is one of those in the set that is part of the M-module and which is also available for use in an exterior operation.

The demon part of an interior operation is created to match a need. It is plausible to suggest that it should be created no later than the time the need for it first becomes tangible, i.e., when the possibility of a constraint violation first becomes foreseeable. This statement is useful, although not as precise as we would like, for there is no clear definition as to when a need becomes tangible. Yet the assertion has intuitive substance. For example, in an ACS.1 planner it is reasonable to say that there is no tangible need for a demon to guard against the need to revise a plan before that plan even exists. In a philosophic sense, this may not be true; the potential can be said to have existed always. In a practical sense, however, there is indeed an external event that precedes any need to consider the possibility of replanning. At worst, this event is the one that first created the plan. It may be a later event such as one that actually enter values into the plan. As a practical matter, there is always an event that first makes evident the need to guard against the violation of a particular constraint. This event can be used to initiate the creation of the appropriate demon.

The event that causes a demon to be created can be an exterior or interior operation. In either case, the actual agent that creates the demon can be the call of the procedure used in the operation. If the operation is an interior one, the use of the procedure as the creative agent allows a demon to be generated as a consequence of awakening other demons that already exist. In any case, creation of the demon is the result of an action by a procedure in the set of procedures belonging to the module.

The precondition of a demon can be tested by any action that has the possibility of pushing the M-module into an interior state. Of course, if action does not actually lead to an interior state, no violation will be found and the demon will not be awakened. Any event that alters a value in the set of values contained in the module, or that enters a value where there was none before, may push the module into an interior state. The procedures that are used to enter or change values should, therefore, include the means of testing the preconditions of any demons that may need to be awakened. Hence we have an unambiguous determination as to when we should test the preconditions of any demons that might be involved.

We do not need to test all demons whenever a value is changed, however. The constraints that are of concern relate values within the M-module. A demon that monitors a particular constraint need only be tested when the values related by that constraint are changed. In general, then, the demons used to enforce the constraints can each be linked to particular values or sets of values--and examined only when a change is made in those values.

A demon is tested within the call of the procedure that creates the possibility that that demon might have to be awakened. The call of that procedure is not completed until all affected demons have been tested and any that were awakened as a result have completed executing their procedures. If the called procedures result in other demons being awakened, the execution of their procedures will also be completed. The procedures that are called nest in a way that prevents the initiation of any other exterior operation while the module remains in an interior state, i.e., while some demon remains awake or while some procedure called by a demon is being executed. The isolation required of an interior state is obtained automatically.

The design of procedures to construct demons is not simple. It is complicated by the fact that we want the constraints to be encoded in the model. The procedures that have the responsibility for constructing demons must therefore consult the model to determine

what demons are needed. These procedures get rather complicated, but it is feasible to program them. Such programming is considerably simplified by the decomposition of the system into the hierarchy of M-modules, which allows the designer to consider various requirements in relative isolation. The development of the whole system is not easy, but it is possible and practical even in problem areas that, taken as a whole, are very complex.

V CONCLUSIONS

The use of M-modules for system design appears to have significant advantages for some applications. The characteristics of the application environment that may make the use of M-modules appropriate include the following:

* The system requires the use of a large and complex body of knowledge--too large and too complex to be easily understood or controlled by the user.

* The knowledge is decomposable in a way that permits constructing the system as a hierarchy of M-modules, each of which can fulfill its responsibilities by using only its own part of the system's knowledge.

* The knowledge used by the system is subject to change, either because the situation itself changes or because of the user's changing view of the problem. It is important that the user be able to tune the system, when needed, to the new situation or requirements.

* Preferably, although not necessarily, the decomposition should be into modules, each of which belongs to one of a small number of classes differing mainly in the labels used and the specific knowledge that is applicable. The procedures required of a module in a class should be reasonably common to the class. It should also be convenient to use a single format for the value sets of any module of a given class. This condition minimizes the amount of programming required to construct the system.

Given these characteristics of the application area, the use of M-modules appears to offer considerable advantage. The following advantages can be cited as significant in certain circumstances:

* The explicit encoding of knowledge in the models keeps the knowledge available for study and modification.

- The decomposition properties make the behavior of the system easier to understand. The resultant partitioning of the knowledge will help the user determine how the knowledge should be modified to fit a changing need.

- The hierarchical structure of the system permits the specification of conditions that assure freedom from deadlock. It also makes it easier to provide reasonable assurance of system stability, even in application areas that may be quite complex and even though general conditions for assuring stability have not been discovered.

- The decomposition properties appear to fit the requirements for loosely coupled distributed-processing systems. It is possible to specify conditions that allow the avoidance of many of the synchronization problems that can otherwise arise in such systems.

- It appears possible to use the M-module concept to exploit the possibilities of multiple processing for obtaining high processing capability combined with high reliability. The explicit encoding of a module's knowledge in a model allows a module to be reconstructed quickly in a new processor to avoid a hardware failure. Other capabilities such as the control and maintenance of checkpoint data are needed to support the management of the fault-tolerant capability. Means for providing the other necessary capabilities and for managing them have not yet been developed. Nevertheless, there do seem to be tangible possibilities for obtaining a very high degree of fault-tolerance.

The concept of M-modules appears to provide a basis for the design of systems with many useful properties. It may prove to be an important concept in designing practical systems for a number of application areas with diverse requirements.

REFERENCES

[1] M. C.Pease, "ACS.1: An Experimental Automated Command Support System," Trans. on Systems, Man and Communications, SMC-8, No. 10, pp 725-735, Oct. 1978.

[2] M. C. Pease, "Planners of ACS.1," Technical Report 15, SRI International, Menlo Park, CA, November 1977.

[3] M. C. Pease, "The Schedulers of ACS.1," Technical Report 14, SRI International, Menlo Park, CA, September 1977.

[4] D. Sagalowicz, "Message Handler of ACS.1," Technical Report 16, SRI International, Menlo Park, CA, January 1979.

[5] E. W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," in Report on Conference on Software Engineering, Randall and Naur, eds., NATO, 1968.

[6] E. W. Dijkstra, "Notes on Structured Programming," published in Structured Programming by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, N. Y., 1972.

[7] E. D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence Vol. 5, No. 2, pp 115-135, Summer 1974.

[8] E. D. Sacerdoti, "A Structure for Plans and Behavior," Artificial Intelligence Center Technical Note 109, Stanford Research Institute, Menlo Park, CA, August 1975.

[9] E. R. Caianiello, "Automata Theory," Academic Press, N. Y., 1966.

[10] D. L. Parnas, "A Technique for Software Module Specification with Examples," Comm. A. C. M., pp 330-336 May 1972.

[11] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Comm. A. C. M., pp 1053-1058, Dec. 1972.

[12] L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "A Formal Methodology for the Design of Operating System Software," published in Current Trends in Programming Methodology, Vol. 1, ed. by R. T. Yeh, Prentice-Hall Inc., Englewood Cliffs, N. J., April 1977.

[13] M. Minsky, "A framework for representing knowledge," Artificial Intelligence Memo No. 306, Massachusetts Institute of Technology, Cambridge, MA, June 1974.

[14] T. Winograd, "Five Lectures on Artificial Intelligence," Stanford Artificial Intelligence Laboratory, Memo No. AIM-246, Stanford, CA, September 1974.

[15] E. W. Dijkstra, "The Structure of the 'THE'-Multiprograming System," Comm. A.C.M. 11, pp 341-346, May 1968.

[16] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," Proc. of the Third International Joint Conference on Artificial Intelligence, pp 235-245, 1973.

DISTRIBUTION LIST

Defense Documentation Center — 12 copies
Cameron Station
Alexandria, Virginia 22314

Office of Naval Research — 2 copies
Information Systems Program
Code 437
Arlington, Virginia 22217

Office of Naval Research — 1 copy
Branch Office, Boston
495 Summer Street
Boston, Massachusetts 02210

Office of Naval Research — 1 copy
Branch Office, Chicago
536 South Clark Street
Chicago, Illinois 60605

Office of Naval Research — 1 copy
Branch Office, Pasadena
1030 East Green Street

New York Area Office — 1 copy
715 Broadway - 5th Floor
New York, New York 10003

Naval Research Laboratory — 6 copies
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. A. L. Slafkosky — 1 copy
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Office of Naval Research — 1 copy
Code 455
Arlington, Virginia 22217

Office of Naval Research — 1 copy
Code 458
Arlington, Virginia 22217

Naval Ocean Systems Center — 1 copy
Advanced Software Technology Division
Code 822
San Diego, California 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Dev. Center
Computation and Mathematics Dept.
Bethesda, Maryland 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS PLANNING BRANCH (OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Director — 1 copy
National Security Agency
Attn: Mr. Glick
Fort George G. Meade, Maryland 20755

Naval Aviation Integrated Logistic Support Center — 1 copy
Code 800
Patuxent River, Maryland 20670

Professor Omar Wing — 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and Computer Science
New York, New York 10027

Mr. M. Culpepper — 1 copy
Code 183
Naval Ship Research and Development Center
Bethesda, Maryland 20084

Mr. D. Jefferson — 1 copy
Code 188
Naval Ship Research and Development Center
Bethesda, Maryland 20084

Robert C. Kolb, Head — 1 copy
Code 824
Tactical Command Control and Navigation Division
Naval Ocean Systems Center
San Diego, California 92152

Defense Mapping Agency Topographic Center — 1 copy
Attn: Advanced Technology Division
Code 41300 (Mr. W. Mullison)
6500 Brookes Lane
Washington, D.C. 20315

Commander, Naval Sea Systems Command — 1 copy
Department of the NAvy
Attn: PMS 30611
Washington, D.C. 20362

Professor Mike Athans — 1 copy
MIT
Dept. of Elec. Eng. & Comp. Science
77 Massachusetts Avenue
Cambridge, Massachusetts 02139

Captain Richard L. Martin — 1 copy
Cmd. Officer, USS Francis Marion
LPA-249
FPO, New York 09051